MAOR

Android Mobile Advertising Orchestrator and Rendering

Install

In the main project build.gradle file, configure the required Maven repositories

- ItaliaOnline GitHub repository - provided by ItaliaOnline to the customers
- Iubenda Maven repository - https://libraries.iubenda.com/android
- Teads Maven repository - https://teads.jfrog.io/artifactory/SDKAndroid-maven-prod
- Liveramp Maven repository - https://sdk-android-prod.launch.liveramp.com
- Jitpack repository - https://jitpack.io

For the ItaliaOnline Maven repository, you need to provide your GitHub credentials in order to access the private repository. You can create a personal access token in GitHub and use it as password, while the username is your GitHub username. The url of the ItaliaOnline GitHub repository is provided by ItaliaOnline to the customers.

For example, for Gradle with Kotlin DSL

```
allprojects {

    repositories {
        mavenCentral()
        google()

        // MAOR repo
        maven {
            url = uri("PLACEHOLDER") // placeholder for the ItaliaOnline
GitHub repository url, provided by ItaliaOnline to the customers
            credentials {
                username = "xxxxx"
                password = "yyyyyyyyyyyyyyyyy"
            }
        }

        // Iubenda repo
        maven {
            url = uri("https://libraries.iubenda.com/android")
        }

        // Teads repo
        maven {
            url = uri("https://teads.jfrog.io/artifactory/SDKAndroid-maven-
prod")
        }

        // Liveramp repo
```

```
        maven {
            url = uri("https://sdk-android-prod.launch.liveramp.com")
        }

        // Jitpack repo
        maven {
            url = uri("https://jitpack.io")
        }
    }
}
```

Then, in the application module build.gradle file, add the latest MAOR dependencies. The maven coordinates are:

- groupId: "it.italiaonline"
- artifactId: "maor"

For example, for Gradle with Kotlin DSL

```
implementation("it.italiaonline:maor:4.4.5")
```

Init library

Init at app startup

In order to init the MAOR library, call in your Application class the `initMaor` static method in Maor class. You can call the init method in the `onCreate` method and it is safe to call it in the MainThread.

```
fun initMaor(
    application: Application,
    mobileAdsTestMode: Boolean,
    maorConfig: MaorConfig,
    maorFeatureFlags: MaorFeatureFlags = MaorFeatureFlags(),
    onInitResult: (MaorInitState) -> Unit,
)
```

The `initMaor` method requires:

- `application`: The `android.app.Application` instance
- `mobileAdsTestMode`: a boolean value for ADV TEST mode
- `maorConfig`: the MaorConfig instance
- `maorFeatureFlags` : a function that returns a MaorFeatureFlags instance (for on-the run config changes)
- `onInitResult`: the init callback

Example

```
class DemoApp : Application() {

    [...]

    override fun onCreate() {
        super.onCreate()

        Maor.instance.initMaor(
            application = this,
            mobileAdsTestMode = true,
            maorConfig = MaorConfig(
                config = UrlConfig.getGlobalConfig(),
                liveRampKey = "fab84c6e-134c-4c1a-bfe2-fa266824c863",
                storeUrl = "https://play.google.com/store/apps/details?
id=your-app-id",
            ),
            maorFeatureFlags = MaorFeatureFlags(
                getYieldbirdFallbackEnabled = {
                    true
                }
            ),
            onInitResult = { result ->
                Timber.i("is Maor initialized = $result")
            }
        )
    }
}
```

Init adv config

After MAOR init, you can load your specific ADV configuration using

```
fun initAdvManager(
    managerConfig: AdvManagerConfig,
    initDispatcher: CoroutineDispatcher = Dispatchers.Main,
    identifier: (AdvIdentifier) -> Unit,
    )
```

Use `AdvManagerConfig.Builder()` builder class in order to setup:

- The ADV and Engine json config url

The `(AdvIdentifier) -> Unit` callbacks returns the internal AdvIdentifier associated to the config provided

For example:

```
Maor.instance.initAdvManager(
    managerConfig = AdvManagerConfig
        .Builder(configuration = ADV_CONFIG_JSON_URL)
        .build()
) { advIdentifier ->
    Timber.d("AdvIdentifier = $advIdentifier")
}
```

Handle CMP permission

The `Maor` singleton class exposes the followint CMP specific functions:

- `hasCmpConsent()`: return TRUE if CMP consent is given, FALSE otherwise
- `getConsentString()`: return che CMP consent string
- `shouldGetConsent()`: return TRUE if the user needs to setup CMP, FALSE otherwise
- `shouldGetConsentIfExpired()`: return TRUE if the user needs to setup CMP, using the `invalidateConsentBefore` iubenda config parameter
- `askCmpConsent(activity: Activity)`: starts the CMP flow
- `cmpClear()`: reset the CMP handler to the initial value
- `editCmpConsent(activity: Activity)`: starts the CMP edit flow
- `setCmpListener(listener: CMPConsentChangeListener)`: configure an instance of `CMPConsentChangeListener` in order to be notified if user changes the CMP consent

```
interface CMPConsentChangeListener {
    fun onConsentChanged()
}
```

- `removeCmpListener()` : remove a previously configured `CMPConsentChangeListener`
- `getConsentTimestamp()`: return the user consent timestamp as a Date object
- `getConsentId()` : return the user consent ID, if it is present
- `isPurposeConsentGivenForPurposeId(id: Int)` : return TRUE if consent is given for a specific purpose Id
- `acceptAllConsents()`: accept all the consents programmatically

Request an ADV

In order to get an ADV, call the `getMaorADV` method in `Maor` singleton class.

```
fun getMaorADV(
    advReference: AdvReference,
    activityWeakReference: WeakReference<Activity>,
    maorCallback: MaorCallback? = null,
)
```

- `advReference`: AdvReference instance that identifies which kind of ADV should be loaded by Maor
- `activityWeakReference`: a weak reference to the activity class
- `maorCallback`: the callback that receives the ADV loading callbacks

```
data class AdvReference(
    val widgetIndex: Int = 0,
    val indexAdUnit: Int = 0,
    val advId: String,
    val identifier: AdvIdentifier,
)
```

- `widgetIndex`: outBrain index, used only if OutBrain is configured
- `indexAdUnit`: the ADUnit index defined in the config.json file
- `advId`: the ADV type configured in the config.json file, for example "top" or "interstitial".
- `identifier`: the AdvIdentifier instance returned by `Maor.instance.initAdvManager`

For example, for the following config.json snippet

```
[...]
{
    "android": [
        "/1111/test_app/top/android/1",
        "/1111/test_app/top/android/2"
    ],
    "ios": [
        "/1111/test_app/top/ios/1",
        "/1111/test_app/top/ios/2"
    ],
    "size": {
        "width": 320,
        "height": 160
    },
    "format": [
        {
            "id": "native"
        }
    ],
    "id": "top"
}
[...]
```

`AdvReference(indexAdUnit = 2, advIdentifier = "top", identifier = identifier)` loads the `/1111/test_app/top/android/2` adUnit

The `MaorCallback` interface is defined as follows:

```
interface MaorCallback {
    fun onMaorAdLoaded(maorAd: MaorAd)
    fun onMaorLoading()
    fun onMaorError(ex: MaorException)
}
```

- `onMaorLoading`: called when Maor starts loading the ADV
- `onMaorAdLoaded`: when AD is loaded: the MaorAd instance passed as arguments contains the ADV data
- `onMaorError`: called when an Exception is thrown during the ADV loading

This is a full example

```
Maor.instance.getMaorADV(
    advReference = AdvReference(
        indexAdUnit = indexAdUnit,
        advId = advId,
        identifier = id,
        widgetIndex = widgetIndex,
    ),
    activityWeakReference = WeakReference(activity),
    maorCallback = object : MaorCallback {
        override fun onMaorAdLoaded(maorAd: MaorAd) {
            Timber.i("[ADV] - END LOADING ADV - maorAd is $maorAd")
        }

        override fun onMaorError(ex: MaorException) {
            Timber.w("[ADV] - END LOADING ADV with error - $ex ")
        }

        override fun onMaorLoading() {
            Timber.d("Loading ADV $advId - adUnit $indexAdUnit")
        }
    }
)
```

Interstitial Prefetch

MAOR loads the interstitial with a prefetch phase, in order to follow the Google GAM guidelines. For example, if you need to load an interstitial when the user opens the "ItemDetailFragment" and then come back to the "MainFragment", you should:

- call the getMaorADV function when the ItemDetailFragment is created , providing the right Interstitial AdvReference
- MAOR in the meantime loads the interstitial, without rendering it
- call `suspend fun renderMaorInterstitial(activityWeakReference: WeakReference<Activity>)` when the user closes the Fragment and goes back to "MainFragment". ** if the interstitial ADV is ready, is now shown to the user ** If the interstitial ADV

is not yet loaded, a "loading spinner" is shown to the user: after 5 reduce loading times, if the interstitial ADV is not ready, this function returns without blocking the user.

Banner prefetch

It is also possible to prefetch a banner-type AD unit by implementing a simple local cache within your application.

For example, if you want to reduce the loading time of a banner displayed in an "ItemDetailFragment", which is called from its own "MainFragment", you can:

- On opening MainFragment, call the ADV via the getMaorADV call with the correct AdvReference instance.
- Store the response in a local in-memory cache
- On opening ItemDetailFragment, request the adv directly from the local cache, instead of making a new network call.

This methodology, which does not violate Google GAM guidelines, can significantly reduce, if not completely eliminate, the loading time of a single ADV.

The local cache must then be invalidated according to your use cases, for example:

- In case of account change, if the application supports multi-account profiling
- In case of CMP preferences change